16 Designing for fault tolerance

This chapter covers

- What is fault-tolerance and why do you need it?
- Using redundancy to remove single points of failure
- Retrying on failure
- Using idempotent operations to achieve retry on failure
- AWS service guarantees

Failure is inevitable: hard disks, networks, power, and so on all fail from time to time. Fault tolerance deals with that problem. A fault-tolerant architecture is built for failure. If a failure occurs, the system isn't interrupted, and it continues to handle requests. If there is single point of failure within your architecture, it is not fault-tolerant. You can achieve fault-tolerance by introducing redundancy into your system and by decoupling the parts of your architecture such that one side does not rely on the uptime of the other.

The services provided by AWS offer different types of failure resilience:

- *No guarantees (single point of failure)*—No requests are served in case of failure.
- *High availability*—In case of failure, it takes some time until requests are served as before.
- *Fault-tolerance*—In case of failure, requests are served as before without any availability issues.

The most convenient way to make your system fault-tolerant is to build the architecture using fault-tolerant blocks. If all blocks are fault-tolerant, the whole system will be fault-tolerant as well. Luckily, many AWS services are fault-tolerant by default. If possible, use them. Otherwise you'll need to deal with the consequences and handle failures yourself.

Unfortunately, one important service isn't fault-tolerant by default: EC2 instances. Virtual machines aren't fault-tolerant. This means an architecture that uses EC2 isn't fault-tolerant by default. But AWS provides the building blocks to deal with that issue. The solution consists of auto-scaling groups, Elastic Load Balancing (ELB), and Simple Queue Service (SQS).

The following services provided by AWS are neither highly available nor fault-tolerant. When using one of these services in your architecture, you are adding a *single point of failure* (SPOF) to your infrastructure. In this case, to achieve fault-tolerance, you need to plan and build for failure as discussed during the rest of the chapter.

- *Amazon Elastic Compute Cloud (EC2) instance*—A single EC2 instance can fail for many reasons: hardware failure, network problems, availability-zone outage, and so on. To achieve high availability or fault-tolerance, use auto-scaling groups to set up a fleet of EC2 instances that serve requests in a redundant way.
- *Amazon Relational Database Service (RDS) single instance*—A single RDS instance could fail for the same reasons than an EC2 instance might fail. Use Multi-AZ mode to achieve high availability.

All the following services are *highly available* (HA) by default. When a failure occurs, the services will suffer from a short downtime but will recover automatically:

- *Elastic Network Interface (ENI)*—A network interface is bound to an AZ, so if this AZ goes down, your network interface will be unavailable as well.
- *Amazon Virtual Private Cloud (VPC) subnet*—A VPC subnet is bound to an AZ, so if this AZ suffers from an outage, your subnet will not be reachable as well. Use multiple subnets in different AZs to remove the dependency on a single AZ.
- *Amazon Elastic Block Store (EBS) volume*—An EBS volume distributes data among multiple machines within an AZ. But if the whole AZ fails, your volume will be unavailable (you won't lose your data though). You can create EBS snapshots from time to time so you can re-create an EBS volume in another AZ.
- *Amazon Relational Database Service (RDS) Multi-AZ instance*—When running in Multi-AZ mode, a short downtime (1 minute) is expected if an issue occurs with the master instance while changing DNS records to switch to the standby instance.

The following services are *fault-tolerant* by default. As a consumer of the service, you won't notice any failures.

- Elastic Load Balancing (ELB), deployed to at least two AZs
- Amazon EC2 security groups
- Amazon Virtual Private Cloud (VPC) with an ACL and a route table
- Elastic IP addresses (EIP)
- Amazon Simple Storage Service (S3)
- Amazon Elastic Block Store (EBS) snapshots
- Amazon DynamoDB
- Amazon CloudWatch
- Auto-scaling groups
- Amazon Simple Queue Service (SQS)
- AWS Elastic Beanstalk—The management service itself, not necessarily your application running within the environment
- AWS OpsWorks—The management service itself, not necessarily your application running within the environment
- AWS CloudFormation
- AWS Identity and Access Management (IAM, not bound to a single region; if you create an IAM user, that user is available in all regions)

Why should you care about fault tolerance? Because in the end, a fault-tolerant system provides the highest quality to your end users. No matter what happens in your system, the user is never affected and can continue to consume entertaining content, buy goods and services, or have conversations with friends. A few years ago, achieving fault tolerance was expensive and complicated, but in AWS, providing fault-tolerant systems is becoming an affordable standard. Nevertheless, building fault-tolerant systems is the supreme discipline of cloud computing, and might be challenging at the beginning.

Chapter requirements

To fully understand this chapter, you need to have read and understood the following concepts:

- EC2 (chapter 3)
- Auto-scaling (chapter 14)
- Elastic Load Balancing (chapter 15)
- Simple Queue Service (chapter 15)

On top of that, the example included in this chapter makes intensive use of the following:

- Elastic Beanstalk (chapter 5)
- DynamoDB (chapter 13)
- Express, a Node.js web application framework

In this chapter, you'll learn everything you need to design a fault-tolerant web application based on EC2 instances (which aren't fault-tolerant by default).

16.1 Using redundant EC2 instances to increase availability

Here are just a few reasons why your virtual machine might fail:

- If the host hardware fails, it can no longer host the virtual machine on top of it.
- If the network connection to/from the host is interrupted, the virtual machine will lose the ability to communicate over network.
- If the host system is disconnected from the power supply, the virtual machine will fail as well.

Additionally the software running inside your virtual machine may also cause a crash:

- If your application contains a memory leak, you'll run out of memory and fail. It may take a day, a month, a year, or more, but eventually it will happen.
- If your application writes to disk and never deletes its data, you'll run out of disk space sooner or later, causing your application to fail.
- Your application may not handle edge cases properly and may instead crash unexpectedly.

Regardless of whether the host system or your application is the cause of a failure, a single EC2 instance is a single point of failure. If you rely on a single EC2 instance, your system will blow up eventually. It's merely a matter of time.

16.1.1 Redundancy can remove a single point of failure

Imagine a production line that makes fluffy cloud pies. Producing a fluffy cloud pie requires several production steps (simplified!):

1. Produce a pie crust.
2. Cool the pie crust.
3. Put the fluffy cloud mass on top of the pie crust.
4. Cool the fluffy cloud pie.
5. Package the fluffy cloud pie.

The current setup is a single production line. The big problem with this setup is that whenever one of the steps crashes, the entire production line must be stopped. Figure 16.1 illustrates the problem when the second step (cooling the pie crust) crashes. The steps that follow no longer work either, because they no longer receive cool pie crusts.

Why not have multiple production lines? Instead of one line, suppose we have three. If one of the lines fails, the other two can still produce fluffy cloud pies for all the hungry customers in the world. Figure 16.2 shows the improvements; the only downside is that we need three times as many machines.

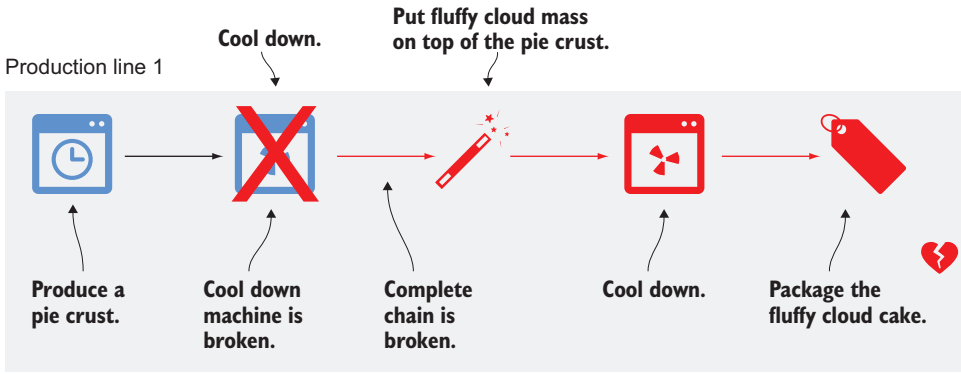
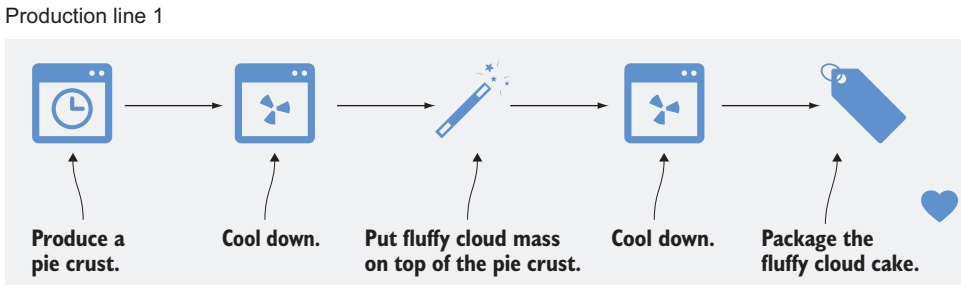
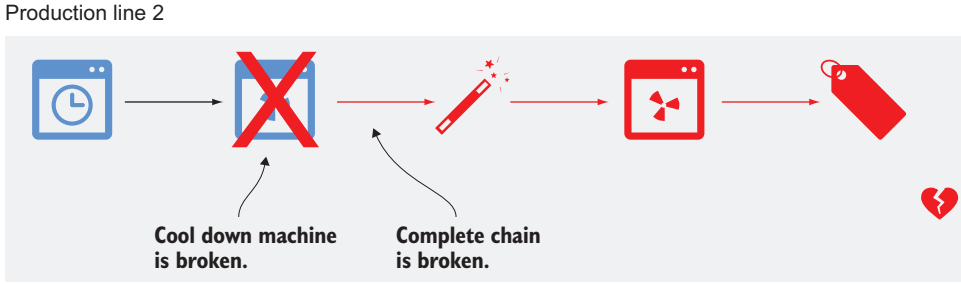


Figure 16.1 A single point of failure affects not only itself, but the entire system.





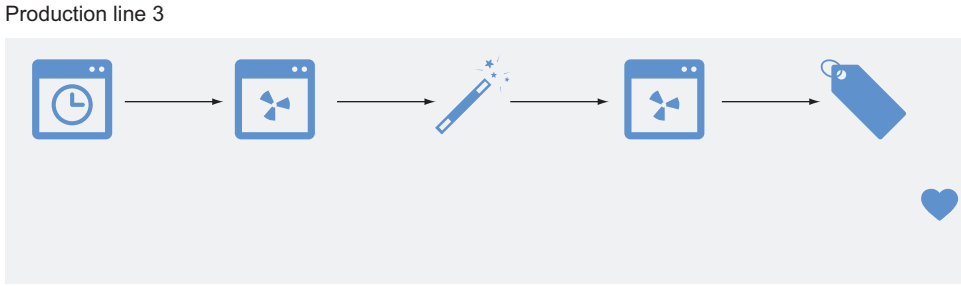


Figure 16.2 Redundancy eliminates single points of failure and makes the system more stable.

The example can be transferred to EC2 instances as well. Instead of having only one EC2 instance running your application, you can have three. If one of those instances fails, the other two will still be able to serve incoming requests. You can also minimize the cost impact of one versus three instances: instead of one large EC2 instance, you can choose three small ones. The problem that arises with a dynamic EC2 instance pool is: how can you communicate with the instances? The answer is *decoupling*: put a load balancer or message queue between your EC2 instances and the requester. Read on to learn how this works.

16.1.2 Redundancy requires decoupling

Figure 16.3 shows how EC2 instances can be made fault-tolerant by using redundancy and synchronous decoupling. If one of the EC2 instances crashes, the Elastic Load Balancer (ELB) stops routing requests to the crashed instances. The auto-scaling group replaces the crashed EC2 instance within minutes, and the ELB begins to route requests to the new instance.

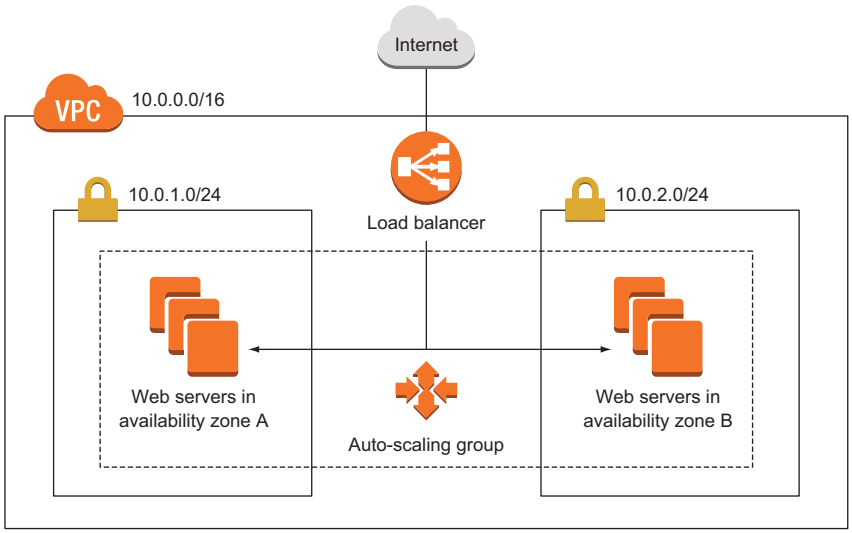


Figure 16.3 Fault-tolerant EC2 instances with an auto-scaling group and an Elastic Load Balancer

Take a second look at figure 16.3 and see what parts are redundant:

- *Availability zones (AZs)*—Two are used. If one AZ suffers from an outage, we still have instances running in the other AZ.
- *Subnets*—A subnet is tightly coupled to an AZ. Therefore we need one subnet in each AZ.
- *EC2 instances*—We have multi-redundancy for EC2 instances. We have multiple instances in one subnet (AZ), and we have instances in two subnets (AZs).

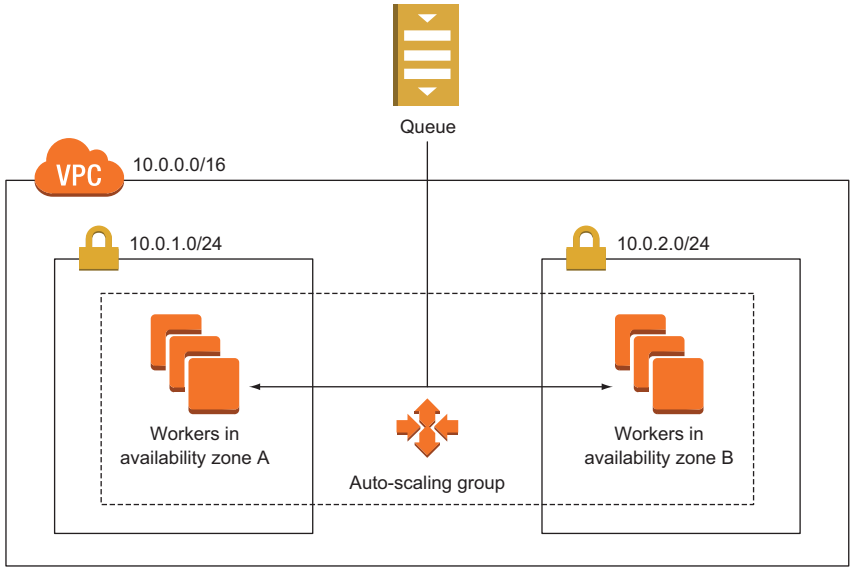


Figure 16.4 Fault-tolerant EC2 instances with an auto-scaling group and SQS

Figure 16.4 shows a fault-tolerant system built with EC2 that uses the power of redundancy and asynchronous decoupling to process messages from an SQS queue.

In both figures, the load balancer and the SQS queue appear only once. This doesn't mean that ELB or SQS are single points of failure; on the contrary, ELB and SQS are both fault-tolerant by default.

16.2 Considerations for making your code fault-tolerant

If you want to achieve fault tolerance, you have to build your application accordingly. You can design fault tolerance into your application by following two suggestions presented in this section.

16.2.1 Let it crash, but also retry

The Erlang programming language is famous for the concept of "let it crash." That means whenever the program doesn't know what to do, it crashes, and someone needs to deal with the crash. Most often people overlook the fact that Erlang is also famous for retrying. Letting it crash without retrying isn't useful—if you can't recover from a crash, your system will be down, which is the opposite of what you want.

You can apply the "let it crash" concept (some people call it "fail fast") to synchronous and asynchronous decoupled scenarios. In a synchronous decoupled scenario, the sender of a request must implement the retry logic. If no response is returned within a certain amount of time, or an error is returned, the sender retries by sending the same request again. In an asynchronous decoupled scenario, things are easier. If a message is consumed but not acknowledged within a certain amount of time, it goes

back to the queue. The next consumer then grabs the message and processes it again. Retrying is built into asynchronous systems by default.

"Let it crash" isn't useful in all situations. If the program wants to respond to the sender that the request contained invalid content, this isn't a reason for letting the server crash: the result will stay the same no matter how often you retry. But if the server can't reach the database, it makes a lot of sense to retry. Within a few seconds, the database may be available again and able to successfully process the retried request.

Retrying isn't that easy. Imagine that you want to retry the creation of a blog post. With every retry, a new entry in the database is created, containing the same data as before. You end up with many duplicates in the database. Preventing this involves a powerful concept that's introduced next: idempotent retry.

16.2.2 Idempotent retry makes fault tolerance possible

How can you prevent a blog post from being added to the database multiple times because of a retry? A naïve approach would be to use the title as primary key. If the primary key is already used, you can assume that the post is already in the database and skip the step of inserting it into the database. Now the insertion of blog posts is *idempotent*, which means no matter how often a certain action is applied, the outcome must be the same. In the current example, the outcome is a database entry.

It continues with a more complicated example. Inserting a blog post is more complicated in reality, as the process might look something like this:

1. Create a blog post entry in the database.
2. Invalidate the cache because data has changed.
3. Post the link to the blog's Twitter feed.

Let's take a close look at each step.

1. CREATING A BLOG POST ENTRY IN THE DATABASE

We covered this step earlier by using the title as a primary key. But this time, we use a universally unique identifier (UUID) instead of the title as the primary key. A UUID like `550e8400-e29b-11d4-a716-446655440000` is a random ID that's generated by the client. Because of the nature of a UUID, it's unlikely that two identical UUIDs will be generated. If the client wants to create a blog post, it must send a request to the ELB containing the UUID, title, and text. The ELB routes the request to one of the backend servers. The backend server checks whether the primary key already exists. If not, a new record is added to the database. If it exists, the insertion continues. Figure 16.5 shows the flow.

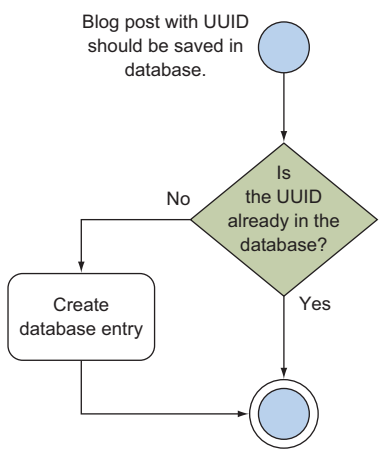


Figure 16.5 Idempotent database insert: creating a blog post entry in the database only if it doesn't already exist

Creating a blog post is a good example of an idempotent operation that is guaranteed by code. You can also use your database to handle this problem. Just send an insert to your database. Three things could happen:

1. *Your database inserts the data.* The operation is successfully completed.
2. *Your database responds with an error because the primary key is already in use.* The operation is successfully completed.
3. *Your database responds with a different error.* The operation crashes.

Think twice about the best way to implement idempotence!

2. Invalidating the cache

This step sends an invalidation message to a caching layer. You don't need to worry about idempotence too much here: it doesn't hurt if the cache is invalidated more often than needed. If the cache is invalidated, then the next time a request hits the cache, the cache won't contain data, and the original source (in this case, the database) will be queried for the result. The result is then put in the cache for subsequent requests. If you invalidate the cache multiple times because of a retry, the worst thing that can happen is that you may need to make a few more calls to your database. That's easy.

3. Posting to the blog's Twitter feed

To make this step idempotent, you need to use some tricks, because you interact with a third party that doesn't support idempotent operations. Unfortunately, no solution will guarantee that you post exactly one status update to Twitter. You can guarantee the creation of at least one (one or more than one) status update, or at most one (one or none) status update. An easy approach could be to ask the Twitter API for the latest status updates; if one of them matches the status update that you want to post, you skip the step because it's already done.

But Twitter is an eventually consistent system: there is no guarantee that you'll see a status update immediately after you post it. Therefore, you can end up having your status update posted multiple times. Another approach would be to save in a database whether you already posted the status update. But imagine saving to the database that you posted to Twitter and then making the request to the Twitter API—but at that moment, the system crashes. Your database will state that the Twitter status update was posted, but in reality it wasn't. You need to make a choice: tolerate a missing status update, or tolerate multiple status updates. Hint: it's a business decision. Figure 16.6 shows the flow of both solutions.

Now it's time for a practical example! You'll design, implement, and deploy a distributed, fault-tolerant web application on AWS. This example will demonstrate how distributed systems work and will combine most of the knowledge in this book.

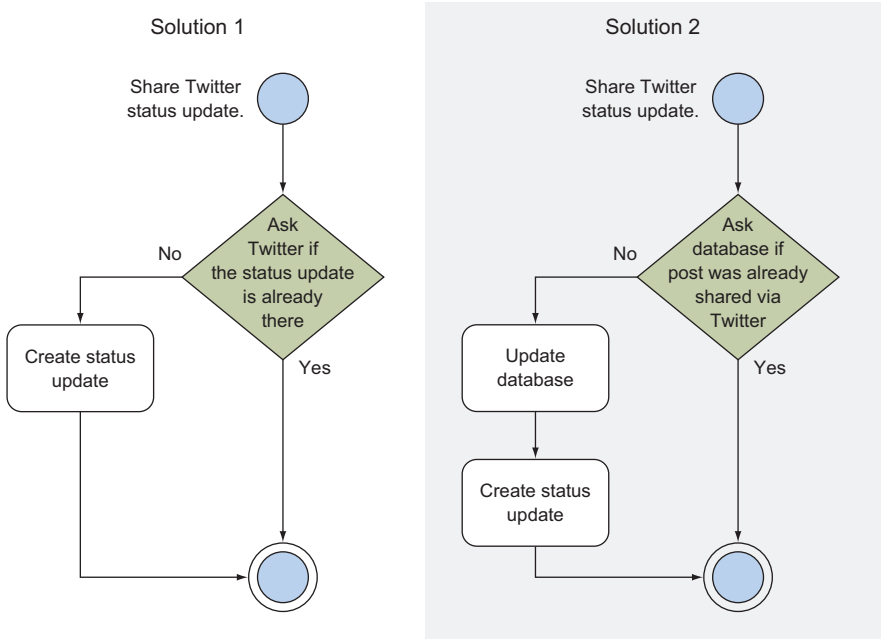


Figure 16.6 Idempotent Twitter status update: only share a status update if it hasn't already been done.

16.3 Building a fault-tolerant web application: Imagery

Before you begin the architecture and design of the fault-tolerant Imagery application, we'll talk briefly about what the application should do. A user should be able to upload an image. This image is then transformed with a sepia filter so that it looks old. The user can then view the sepia image. Figure 16.7 shows the process.

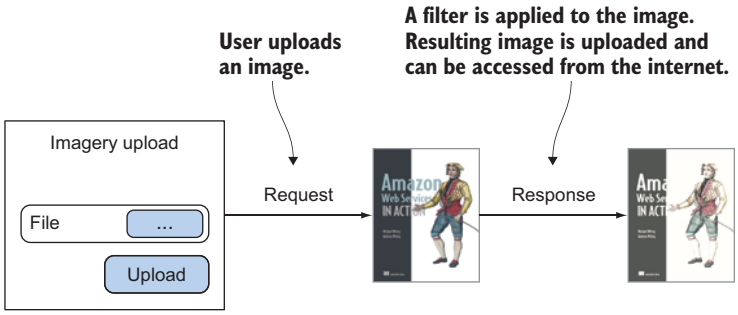


Figure 16.7 The user uploads an image to Imagery, where a filter is applied.

The problem with the process shown in figure 16.7 is that it's synchronous. If the web server dies during request and response, the user's image won't be processed. Another problem arises when many users want to use the Imagery app: the system

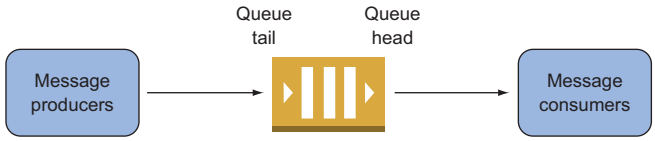


Figure 16.8 Producers send messages to a message queue while consumers read messages.

becomes busy and may slow down or stop working. Therefore the process should be turned into an asynchronous one. Chapter 15 introduced the idea of asynchronous decoupling by using an SQS message queue, as shown in figure 16.8.

When designing an asynchronous process, it's important to keep track of the process. You need some kind of identifier for it. When a user wants to upload an image, the user creates a process first. This returns a unique ID. With that ID, the user can upload an image. If the image upload is finished, the worker begins to process the image in the background. The user can look up the process at any time with the process ID. While the image is being processed, the user can't see the sepia image. But as soon as the image is processed, the lookup process returns the sepia image. Figure 16.9 shows the asynchronous process.

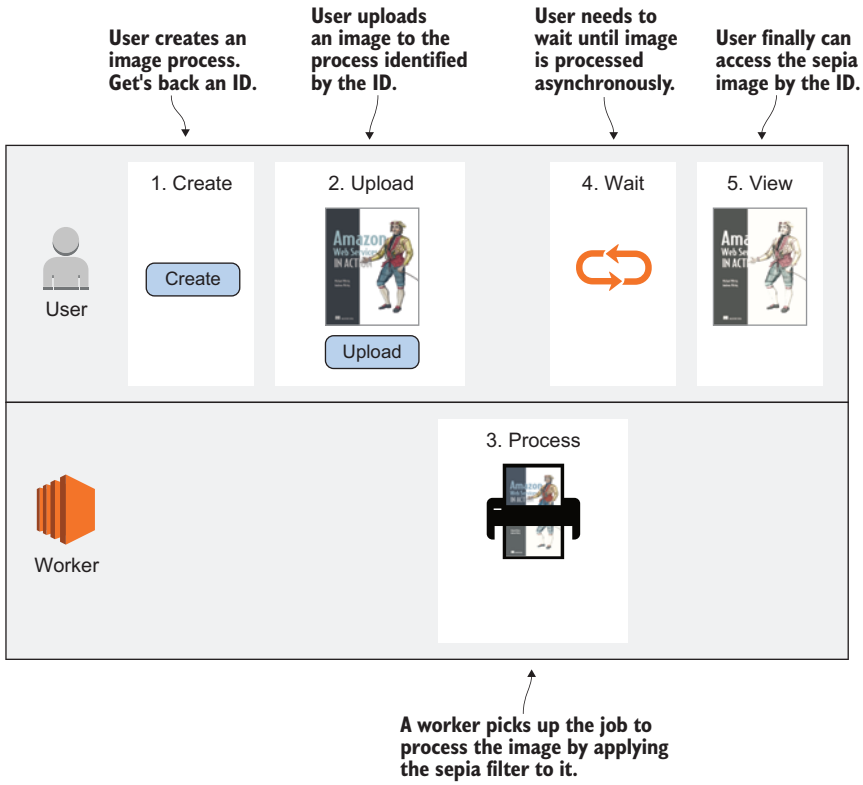


Figure 16.9 The user asynchronously uploads an image to Imagery, where a filter is applied.

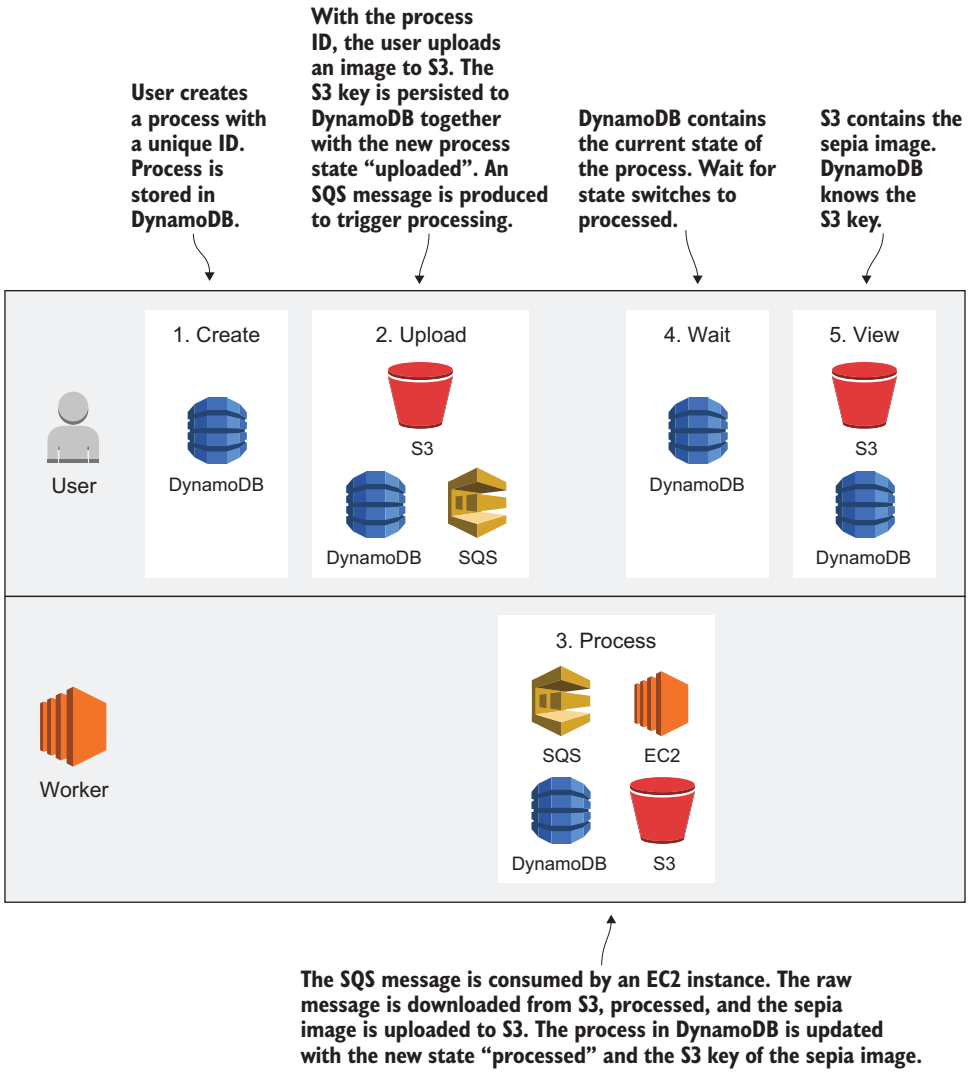


Figure 16.10 Combining AWS services to implement the asynchronous Imagery process

Now that you have an asynchronous process, it's time to map that process to AWS services. Keep in mind that most services on AWS are fault-tolerant by default, so it makes sense to pick them whenever possible. Figure 16.10 shows one way of doing it.

To make things as easy as possible, all the actions will be accessible via a REST API, which will be provided by EC2 instances. In the end, EC2 instances will provide the process and make calls to all the AWS services shown in figure 16.10.

You'll use many AWS services to implement the Imagery application. Most of them are fault-tolerant by default, but EC2 isn't. You'll deal with that problem using an idempotent state machine, as introduced in the next section.

Example is 100% covered by the Free Tier

The example in this chapter is totally covered by the Free Tier. As long as you don't run the example longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the example within a few days, because you'll clean up your account at the end of the section.

16.3.1 The idempotent state machine

An idempotent state machine sounds complicated. We'll take some time to explain it because it's the heart of the Imagery application. Let's look at what a *state machine* is and what *idempotent* means in this context.

THE FINITE STATE MACHINE

A finite state machine has at least one start state and one end state. Between the start and the end state, the state machine can have many other states. The machine also defines transitions between states. For example, a state machine with three states could look like this:

```
(A) -> (B) -> (C).
```

This means:

- State A is the start state.
- There is a transition possible from state A to B.
- There is a transition possible from state B to C.
- State C is the end state.

But there is no transition possible between (A) -> (C) or (B) -> (A). With this in mind, we apply the theory to our Imagery example. The Imagery state machine could look like this:

```
(Created) -> (Uploaded) -> (Processed)
```

Once a new process (state machine) is created, the only transition possible is to `Uploaded`. To make this transition happen, you need the S3 key of the uploaded raw image. So the transition between `Created` -> `Uploaded` can be defined by the function `uploaded(s3Key)`. Basically, the same is true for the transition `Uploaded` -> `Processed`. This transition can be done with the S3 key of the sepia image: `processed(s3Key)`.

Don't be confused by the fact that the upload and the image filter processing don't appear in the state machine. These are the basic actions that happen, but we're only interested in the results; we don't track the progress of the actions. The process isn't aware that 10% of the data has been uploaded or 30% of the image processing is done. It only cares whether the actions are 100% done. You can probably imagine a

bunch of other states that could be implemented, but we're skipping that for the purpose of simplicity in this example: resized and shared are just two examples.

IDEMPOTENT STATE TRANSITIONS

An idempotent state transition must have the same result no matter how often the transition takes place. If you can make sure that your state transitions are idempotent, you can do a simple trick: in case of a failure during transitioning, you retry the entire state transition.

Let's look at the two state transitions you need to implement. The first transition `Created -> Uploaded` can be implemented like this (pseudo code):

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== 'Created') {
    throw new Error('transition not allowed')
  }
  DynamoDB.updateItem(processId, {'state': 'Uploaded', 'rawS3Key': s3Key})
  SQS.sendMessage({'processId': processId, 'action': 'process'});
}
```

The problem with this implementation is that it's not idempotent. Imagine that `SQS.sendMessage` fails. The state transition will fail, so you retry. But the second call to `uploaded(s3Key)` will throw a "transition not allowed" error because `DynamoDB.updateItem` was successful during the first call.

To fix that, you need to change the `if` statement to make the function idempotent:

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== 'Created' && process.state !== 'Uploaded') {
    throw new Error('transition not allowed')
  }
  DynamoDB.updateItem(processId, {'state': 'Uploaded', 'rawS3Key': s3Key})
  SQS.sendMessage({'processId': processId, 'action': 'process'});
}
```

If you retry now, you'll make multiple updates to Dynamo, which doesn't hurt. And you may send multiple SQS messages, which also doesn't hurt, because the SQS message consumer must be idempotent as well. The same applies to the transition `Uploaded -> Processed`.

Next, you'll begin to implement the Imagery server.

16.3.2 Implementing a fault-tolerant web service

We'll split the Imagery application into two parts: the web servers and the workers. As illustrated in figure 16.11, the web servers provide the REST API to the user, and the workers process images.

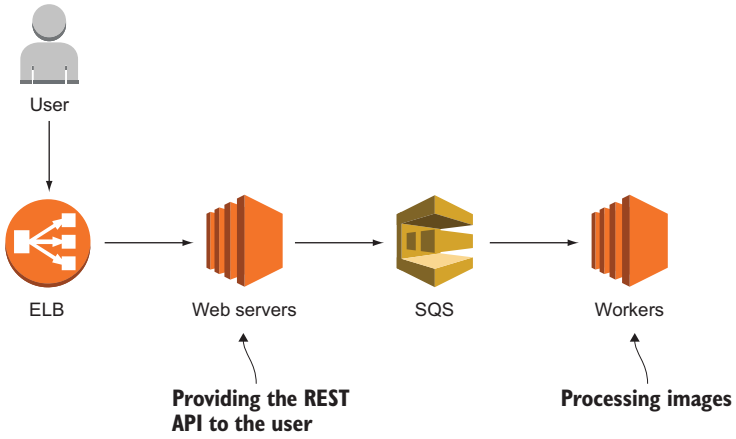


Figure 16.11 The Imagery application consists of two parts: the web servers and the workers.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. Imagery is located in /chapter16/.

The REST API will support the following routes:

- `POST /image`—A new image process is created when executing this route.
- `GET /image/:id`—This route returns the state of the process specified with the path parameter `:id`.
- `POST /image/:id/upload`—This route offers a file upload for the process specified with the `path` parameter `:id`.

To implement the web server, you'll again use Node.js and the Express web application framework. You'll use the Express framework, but don't feel intimidated, as you won't need to understand it in depth to follow along.

Setting up the web server project

As always, you need some boilerplate code to load dependencies, initial AWS endpoints, and things like that.

Listing 16.1 Initializing the Imagery server (server/server.js)

```
var express = require('express');          <-- Load Node.js modules (dependencies)
var bodyParser = require('body-parser');
var AWS = require('aws-sdk');
var uuidv4 = require('uuid/v4');
var multiparty = require('multiparty');

var db = new AWS.DynamoDB({                <-- Creates a DynamoDB endpoint
```

```
  'region': 'us-east-1'
});
var sqs = new AWS.SQS({                    <-- Creates an SQS endpoint
  'region': 'us-east-1'
});
var s3 = new AWS.S3({                      <-- Creates an S3 endpoint
  'region': 'us-east-1'
});

var app = express();                       <-- Creates an Express application
app.use(bodyParser.json());                    Tells Express to parse the request bodies

// [...]

app.listen(process.env.PORT || 8080, function() {      <-- Starts Express on the port defined by the environment variable PORT, or defaults to 8080
  console.log('Server started. Open http://localhost:'
➥ + (process.env.PORT || 8080) + ' with browser.');
});
```

Don't worry too much about the boilerplate code; the interesting parts will follow.

CREATING A NEW IMAGERY PROCESS

To provide a REST API to create image processes, a fleet of EC2 instances will run Node.js code behind a load balancer. The image processes will be stored in DynamoDB. Figure 16.12 shows the flow of a request to create a new image process.

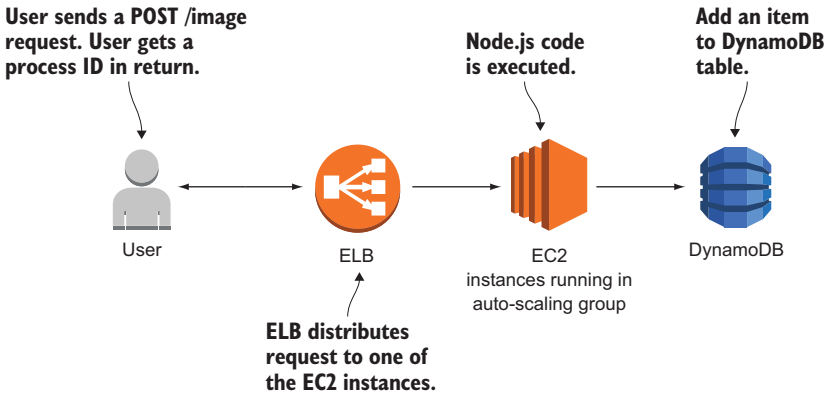
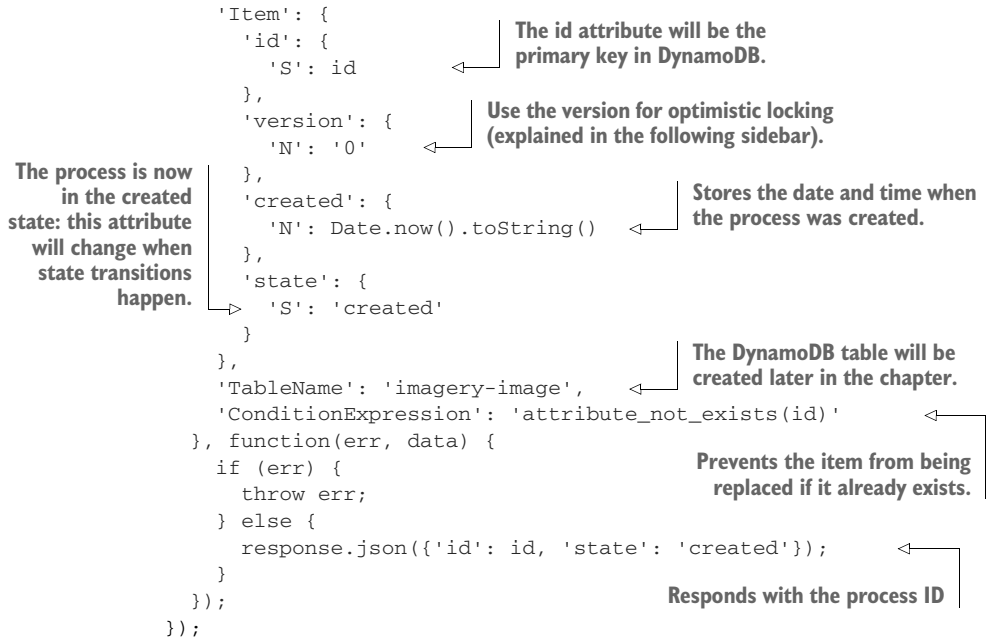


Figure 16.12 Creating a new image process in Imagery

You'll now add a route to the Express application to handle `POST /image` requests, as shown here.

Listing 16.2 Imagery server: `POST /image` creates an image process (server/server.js)

```
app.post('/image', function(request, response) {      <-- Registers the route with Express
    var id = uuidv4();                                <-- Creates a unique ID for the process
    db.putItem({   <-- Invokes the putItem operation on DynamoDB
```



A new process can now be created.

Optimistic locking

To prevent multiple updates to an DynamoDB item, you can use a trick called *optimistic locking*. When you want to update an item, you must specify which version you want to update. If that version doesn't match the current version of the item in the database, your update will be rejected. Keep in mind that optimistic locking is your responsibility, not a default available in DynamoDB. DynamoDB only provides the features to implement optimistic locking.

Imagine the following scenario. An item is created in version 0. Process A looks up that item (version 0). Process B also looks up that item (version 0). Now process A wants to make a change by invoking the `updateItem` operation on DynamoDB. Therefore process A specifies that the expected version is 0. DynamoDB will allow that modification, because the version matches; but DynamoDB will also change the item's version to 1 because an update was performed. Now process B wants to make a modification and sends a request to DynamoDB with the expected item version 0. DynamoDB will reject that modification because the expected version doesn't match the version DynamoDB knows of, which is 1.

To solve the problem for process B, you can use the same trick introduced earlier: retry. Process B will again look up the item, now in version 1, and can (you hope) make the change.

(continued)

There is one problem with optimistic locking: If many modifications happen in parallel, a lot of overhead is created because of many retries. But this is only a problem if you expect a lot of concurrent writes to a single item, which can be solved by changing the data model. That's not the case in the Imagery application. Only a few writes are expected to happen for a single item: optimistic locking is a perfect fit to make sure you don't have two writes where one overrides changes made by another.

The opposite of *optimistic locking* is *pessimistic locking*. A pessimistic lock strategy can be implemented by using a semaphore. Before you change data, you need to lock the semaphore. If the semaphore is already locked, you wait until the semaphore becomes free again.

The next route you need to implement is to look up the current state of a process.

LOOKING UP AN IMAGERY PROCESS

You'll now add a route to the Express application to handle `GET /image/:id` requests. Figure 16.13 shows the request flow.

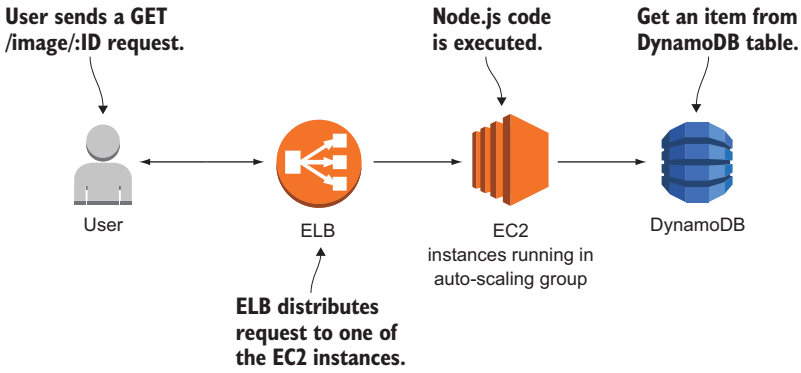


Figure 16.13 Looking up an image process in Imagery to return its state

Express will take care of the path parameter `:id`; Express will provide it within `request.params.id`. The implementation needs to get an item from DynamoDB based on the path parameter ID.

Listing 16.3 `GET /image/:id` looks up an image process (server/server.js)

```
function mapImage = function(item) {
  return {
    'id': item.id.S,
    'version': parseInt(item.version.N, 10),
    'state': item.state.S,
    'rawS3Key': // [...]
    'processedS3Key': // [...]
```

Helper function to map a DynamoDB result to a JavaScript object

```
    'processedImage': // [...]
  };
};

function getImage(id, cb) {
  db.getItem({                          <-- Invokes the getItem operation on DynamoDB
    'Key': {
      'id': {                           <-- id is the partition key.
        'S': id
      }
    },
    'TableName': 'imagery-image'
  }, function(err, data) {
    if (err) {
      cb(err);
    } else {
      if (data.Item) {
        cb(null, lib.mapImage(data.Item));
      } else {
        cb(new Error('image not found'));
      }
    }
  });
}

app.get('/image/:id', function(request, response) {   <-- Registers the route with Express
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      response.json(image);             <-- Responds with the image process
    }
  });
});
```

The only thing missing is the upload part, which comes next.

Uploading an image

Uploading an image via `POST` request requires several steps:

1. Upload the raw image to S3.
2. Modify the item in DynamoDB.
3. Send an SQS message to trigger processing.

Figure 16.14 shows this flow.

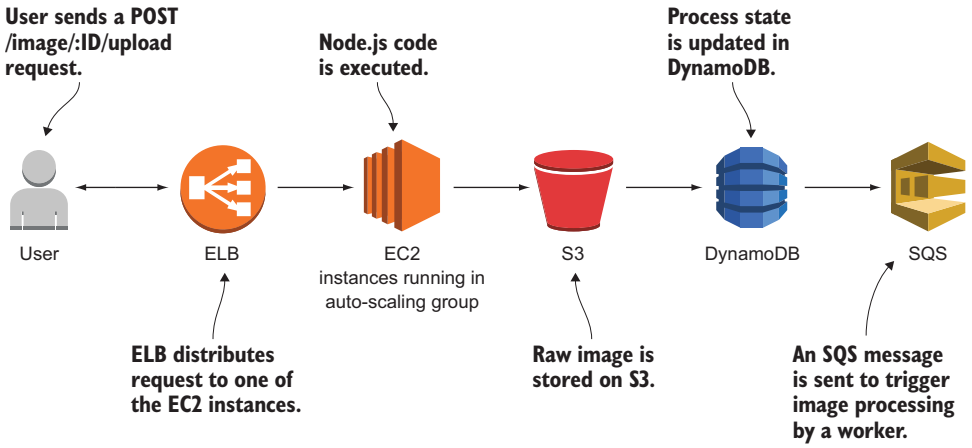


Figure 16.14 Uploading a raw image to Imagery and trigger image processing

Ths listing shows the implementation of these steps.

Listing 16.4 `POST /image/:id/upload` uploads an image (server/server.js)

```
function uploadImage(image, part, response) {
  var rawS3Key = 'upload/' + image.id + '-' + Date.now();
  s3.putObject({
    'Bucket': process.env.ImageBucket,
    'Key': rawS3Key,
    'Body': part,
    'ContentLength': part.byteCount
  }, function(err, data) {
    if (err) {
      throw err;
    } else {
      db.updateItem({
        'Key': {
          'id': {
            'S': image.id
          }
        },
        'UpdateExpression': 'SET #s=:newState,
➥ version=:newVersion, rawS3Key=:rawS3Key',
        'ConditionExpression': 'attribute_exists(id)
➥ AND version=:oldVersion
➥ AND #s IN (:stateCreated, :stateUploaded)',
        'ExpressionAttributeNames': {
          '#s': 'state'
        },
        'ExpressionAttributeValues': {
          ':newState': {
            'S': 'uploaded'
          },
          ':oldVersion': {
```

Creates a key for the S3 object

Calls the S3 API to upload an object

The S3 bucket name is passed in as an environment variable (the bucket will be created later in the chapter).

The body is the uploaded stream of data.

Calls the DynamoDB API to update an object.

Updates the state, version, and raw S3 key

Updates only when item exists. Version equals the expected version, and state is one of those allowed.

```
            'N': image.version.toString()
          },
          ':newVersion': {
            'N': (image.version + 1).toString()
          },
          ':rawS3Key': {
            'S': rawS3Key
          },
          ':stateCreated': {
            'S': 'created'
          },
          ':stateUploaded': {
            'S': 'uploaded'
          }
        },
        'ReturnValues': 'ALL_NEW',
        'TableName': 'imagery-image'
      }, function(err, data) {
        if (err) {
          throw err;
        } else {
          sqs.sendMessage({
            'MessageBody': JSON.stringify(
              {'imageId': image.id, 'desiredState': 'processed'}
            ),
            'QueueUrl': process.env.ImageQueue,
          }, function(err) {
            if (err) {
              throw err;
            } else {
              response.redirect('/#view=' + image.id);
              response.end();
            }
          });
        }
      });
    }
  });
}

app.post('/image/:id/upload', function(request, response) {
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      var form = new multiparty.Form();
      form.on('part', function(part) {
        uploadImage(image, part, response);
      });
      form.parse(request);
    }
  });
});
```

Calls the SQS API to publish a message

Creates the message body containing the image's ID and the desired state

The queue URL is passed in as an environment variable.

Registers the route with Express

We are using the multiparty module to handle multi-part uploads.

The server side is finished. Next you'll continue to implement the processing part in the Imagery worker. After that, you can deploy the application.

16.3.3 Implementing a fault-tolerant worker to consume SQS messages

The Imagery worker does the asynchronous stuff in the background: processing images by applying a sepia filter. The worker handles consuming SQS messages and processing images. Fortunately, consuming SQS messages is a common task that is solved by Elastic Beanstalk, which you'll use later to deploy the application. Elastic Beanstalk can be configured to listen to SQS messages and execute an HTTP `POST` request for every message. In the end, the worker implements a REST API that is invoked by Elastic Beanstalk. To implement the worker, you'll again use Node.js and the Express framework.

SETTING UP THE SERVER PROJECT

As always, you need some boilerplate code to load dependencies, initial AWS endpoints, and so on.

Listing 16.5 Initializing the Imagery worker (worker/worker.js)

```
var express = require('express');
var bodyParser = require('body-parser');
var AWS = require('aws-sdk');
var assert = require('assert-plus');
var Caman = require('caman').Caman;
var fs = require('fs');

var db = new AWS.DynamoDB({
  'region': 'us-east-1'
});
var s3 = new AWS.S3({
  'region': 'us-east-1'
});

var app = express();
app.use(bodyParser.json());

app.get('/', function(request, response) {
  response.json({});
});

// [...]

app.listen(process.env.PORT || 8080, function() {
  console.log('Worker started on port ' + (process.env.PORT || 8080));
});
```

Loads Node.js modules (dependencies)

Creates a DynamoDB endpoint

Creates an S3 endpoint

Creates an Express application

Registers a route for health checks that returns an empty object

Starts Express on a port defined by the environment variable PORT, or defaults to 8080

The Node.js module `caman` is used to create sepia images. You'll wire that up next.

HANDLING SQS MESSAGES AND PROCESSING THE IMAGE

The SQS message to trigger the image processing is handled in the worker. Once a message is received, the worker starts to download the raw image from S3, applies the sepia filter, and uploads the processed image back to S3. After that, the process state in DynamoDB is modified. Figure 16.15 shows the steps.

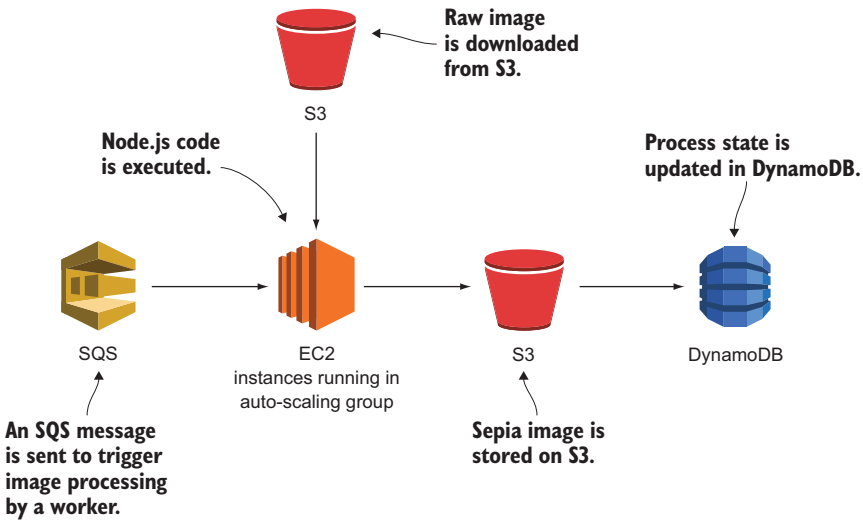


Figure 16.15 Processing a raw image to upload a sepia image to S3

Instead of receiving messages directly from SQS, you'll take a shortcut. Elastic Beanstalk—the deployment tool you'll use—provides a feature that consumes messages from a queue and invokes a HTTP `POST` request for every message. You configure the `POST` request to be made to the resource `/sqs`.

Listing 16.6 Imagery worker: `POST /sqs` handles SQS messages (worker/worker.js)

```
function processImage(image, cb) {
  var processedS3Key = 'processed/' + image.id + '-' + Date.now() + '.png';
  // download raw image from S3
  // process image
  // upload sepia image to S3
  cb(null, processedS3Key);
}

function processed(image, request, response) {
  processImage(image, function(err, processedS3Key) {
    if (err) {
      throw err;
    } else {
      db.updateItem({
```

The implementation of processImage isn't shown here; you can find it in the book's source code folder.

Invokes the updateItem operation on DynamoDB

```
        'Key': {
          'id': {
            'S': image.id
          }
        },
        'UpdateExpression': 'SET #s=:newState,
➥ version=:newVersion, processedS3Key=:processedS3Key',
        'ConditionExpression': 'attribute_exists(id)
➥ AND version=:oldVersion
➥ AND #s IN (:stateUploaded, :stateProcessed)',
        'ExpressionAttributeNames': {
          '#s': 'state'
        },
        'ExpressionAttributeValues': {
          ':newState': {
            'S': 'processed'
          },
          ':oldVersion': {
            'N': image.version.toString()
          },
          ':newVersion': {
            'N': (image.version + 1).toString()
          },
          ':processedS3Key': {
            'S': processedS3Key
          },
          ':stateUploaded': {
            'S': 'uploaded'
          },
          ':stateProcessed': {
            'S': 'processed'
          }
        },
        'ReturnValues': 'ALL_NEW',
        'TableName': 'imagery-image'
      }, function(err, data) {
        if (err) {
          throw err;
        } else {
          response.json(lib.mapImage(data.Attributes));
        }
      });
    }
  });
}

app.post('/sqs', function(request, response) {
  assert.string(request.body.imageId, 'imageId');
  assert.string(request.body.desiredState, 'desiredState');
  getImage(request.body.imageId, function(err, image) {

    if (err) {
      throw err;
    } else {
      if (request.body.desiredState === 'processed') {
```

Updates the state, version, and processed S3 key

Updates only when an item exists, version equals the expected version, and state is one of those allowed.

Responds with the process's new state

Registers the route with Express

The implementation of getImage is the same as on the server.

```
        processed(image, request, response);
      } else {
        throw new Error("unsupported desiredState");
      }
    }
  });
});
```

Invokes the processed function if the SQS message's desiredState equals "processed".

If the `POST /sqs` route responds with a 2XX HTTP status code, Elastic Beanstalk considered the message delivery successful and deletes the message from the queue. Otherwise, the message is redelivered.

Now you can process the SQS message to transform the raw image and upload the sepia version to S3. The next step is to deploy all that code to AWS in a fault-tolerant way.

16.3.4 Deploying the application

As mentioned previously, you'll use Elastic Beanstalk to deploy the server and the worker. You'll use CloudFormation to do so. This may sound strange, because you're using an automation tool to use another automation tool. But CloudFormation does a bit more than just deploying two Elastic Beanstalk applications. It defines the following:

- The S3 bucket for raw and processed images
- The DynamoDB table `imagery-image`
- The SQS queue and dead-letter queue
- IAM roles for the server and worker EC2 instances
- Elastic Beanstalk applications for the server and worker

It takes quite a while to create that CloudFormation stack; that's why you should do so now. After you've created the stack, we'll look at the template. After that, the stack should be ready to use.

To help you deploy Imagery, we created a CloudFormation template located at http://mng.bz/Z33C. Create a stack based on that template. The stack output `EndpointURL` returns the URL that you can access from your browser to use Imagery. Here's how to create the stack from the terminal:

```
$ aws cloudformation create-stack --stack-name imagery \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter16/template.yaml \
➥ --capabilities CAPABILITY_IAM
```

Now let's look at the CloudFormation template.

Deploying S3, DynamoDB, and SQS

Listing 16.7 describes the S3 bucket, DynamoDB table, and SQS queue.

Listing 16.7 Imagery CloudFormation template: S3, DynamoDB, and SQS

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 16'
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'
    Default: mykey
Resources:
  Bucket:
    Type: 'AWS::S3::Bucket'
    Properties:
      BucketName: !Sub 'imagery-${AWS::AccountId}'
      WebsiteConfiguration:
        ErrorDocument: error.html
        IndexDocument: index.html
  Table:
    Type: 'AWS::DynamoDB::Table'
    Properties:
      AttributeDefinitions:
      - AttributeName: id
        AttributeType: S
      KeySchema:
      - AttributeName: id
        KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 1
        WriteCapacityUnits: 1
      TableName: 'imagery-image'
  SQSDLQueue:
    Type: 'AWS::SQS::Queue'
    Properties:
      QueueName: 'message-dlq'
  SQSQueue:
    Type: 'AWS::SQS::Queue'
    Properties:
      QueueName: message
      RedrivePolicy:
        deadLetterTargetArn: !Sub '${SQSDLQueue.Arn}'
        maxReceiveCount: 10
# [...]
Outputs:
  EndpointURL:
    Value: !Sub 'http://${EBServerEnvironment.EndpointURL}'
    Description: Load Balancer URL
```

S3 bucket for uploaded and processed images, with web hosting enabled

The bucket name contains the account ID to make the name unique.

DynamoDB table containing the image processes

The id attribute is used as the partition key.

SQS queue that receives messages that can't be processed

SQS queue to trigger image processing

If a message is received more than 10 times, it's moved to the dead-letter queue.

Visit the output with your browser to use Imagery.

The concept of a *dead-letter queue* (DLQ) needs a short introduction here as well. If a single SQS message can't be processed, the message becomes visible again on the queue for other workers. This is called a *retry*. But if for some reason every retry fails (maybe you have a bug in your code), the message will reside in the queue forever and may waste a lot of resources because of all the retries. To avoid this, you can configure a dead-letter

queue. If a message is retried more than a specific number of times, it's removed from the original queue and forwarded to the DLQ. The difference is that no worker listens for messages on the DLQ. But you should create a CloudWatch alarm that triggers if the DLQ contains more than zero messages, because you need to investigate this problem manually by looking at the message in the DLQ.

Now that the basic resources have been designed, let's move on to the more specific resources.

IAM roles for server and worker EC2 instances

Remember that it's important to only grant the privileges that are necessary. All server instances must be able to do the following:

- `sqs:SendMessage` to the SQS queue created in the template to trigger image processing
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (you can further limit writes to the `upload/` key prefix)
- `dynamodb:GetItem`, `dynamodb:PutItem`, and `dynamodb:UpdateItem` to the DynamoDB table created in the template
- `cloudwatch:PutMetricData`, which is an Elastic Beanstalk requirement
- `s3:Get*`, `s3:List*`, and `s3:PutObject`, which is an Elastic Beanstalk requirement

All worker instances must be able to do the following:

- `sqs:ChangeMessageVisibility`, `sqs:DeleteMessage`, and `sqs:ReceiveMessage` to the SQS queue created in the template
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (you can further limit writes to the `processed/` key prefix)
- `dynamodb:GetItem` and `dynamodb:UpdateItem` to the DynamoDB table created in the template
- `cloudwatch:PutMetricData`, which is an Elastic Beanstalk requirement
- `s3:Get*`, `s3:List*`, and `s3:PutObject`, which is an Elastic Beanstalk requirement

If you don't feel comfortable with IAM roles, take a look at the book's code repository on GitHub at https://github.com/AWSinAction/code2. The template with IAM roles can be found in /chapter16/template.yaml.

Now it's time to design the Elastic Beanstalk applications.

Elastic Beanstalk for the server

First off, a short refresher on Elastic Beanstalk, which we touched on in chapter 5. Elastic Beanstalk consists of these elements:

- An *application* is a logical container. It contains versions, environments, and configurations. To use AWS Elastic Beanstalk in a region, you have to create an application first.
- A *version* contains a specific release of your application. To create a new version, you have to upload your executables (packed into an archive) to S3. A version is basically a pointer to this archive of executables.

- A *configuration template* contains your default configuration. You can manage the configuration of your application (such as the port your application listens on) as well as the configuration of the environment (such as the size of the virtual server) with your custom configuration template.
- An *environment* is the place where AWS Elastic Beanstalk executes your application. It consists of a version and the configuration. You can run multiple environments for one application by using the versions and configurations multiple times.

Figure 16.16 shows the parts of an Elastic Beanstalk application.

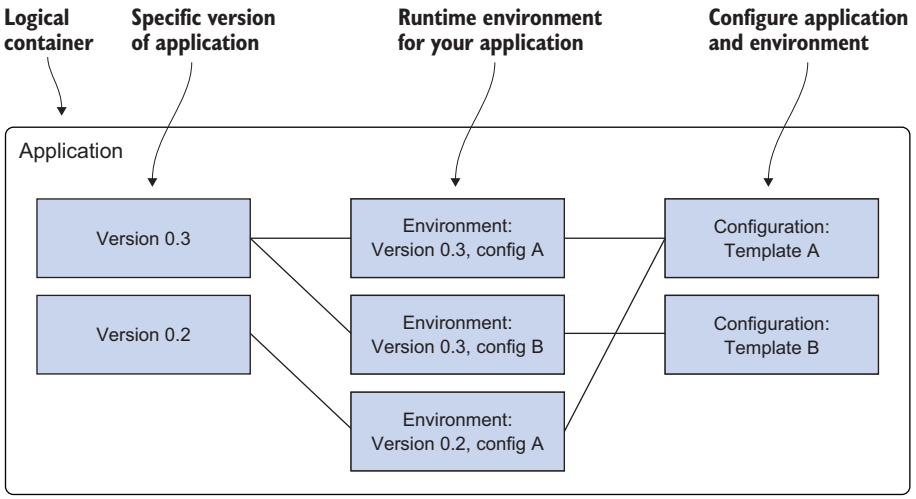


Figure 16.16 An AWS Elastic Beanstalk application consists of versions, environments, and configurations.

Now that you've refreshed your memory, let's look at the Elastic Beanstalk application that deploys the Imagery server.

Listing 16.8 Imagery CloudFormation template: Elastic Beanstalk for the server

```
EBServerApplication:                                    Describes the server
  Type: 'AWS::ElasticBeanstalk::Application'            application container
  Properties:
    ApplicationName: 'imagery-server'
    Description: 'Imagery server: AWS in Action: chapter 16'
EBServerConfigurationTemplate:
  Type: 'AWS::ElasticBeanstalk::ConfigurationTemplate'
  Properties:
    ApplicationName: !Ref EBServerApplication
    Description: 'Imagery server: AWS in Action: chapter 16'
    SolutionStackName:                                  Uses Amazon Linux 2017.09
➥ '64bit Amazon Linux 2017.09 v4.4.0 running Node.js'  running Node.js 6.11.5 per default
    OptionSettings:
    - Namespace: 'aws:autoscaling:asg'                  Minimum of two EC2 instances for
      OptionName: 'MinSize'                             fault-tolerance
      Value: '2'                            )
```

```
      - Namespace: 'aws:autoscaling:launchconfiguration'
        OptionName: 'EC2KeyName'
        Value: !Ref KeyName
      - Namespace: 'aws:autoscaling:launchconfiguration'
        OptionName: 'IamInstanceProfile'
        Value: !Ref ServerInstanceProfile
      - Namespace: 'aws:elasticbeanstalk:container:nodejs'
        OptionName: 'NodeCommand'
        Value: 'node server.js'
      - Namespace: 'aws:elasticbeanstalk:application:environment'
        OptionName: 'ImageQueue'
        Value: !Ref SQSQueue
      - Namespace: 'aws:elasticbeanstalk:application:environment'
        OptionName: 'ImageBucket'
        Value: !Ref Bucket
      - Namespace: 'aws:elasticbeanstalk:container:nodejs:staticfiles'
        OptionName: '/public'
        Value: '/public'
EBServerApplicationVersion:
  Type: 'AWS::ElasticBeanstalk::ApplicationVersion'
  Properties:
    ApplicationName: !Ref EBServerApplication
    Description: 'Imagery server: AWS in Action: chapter 16'
    SourceBundle:
      S3Bucket: 'awsinaction-code2'
      S3Key: 'chapter16/build/server.zip'
EBServerEnvironment:
  Type: 'AWS::ElasticBeanstalk::Environment'
  Properties:
    ApplicationName: !Ref EBServerApplication
    Description: 'Imagery server: AWS in Action: chapter 16'
    TemplateName: !Ref EBServerConfigurationTemplate
    VersionLabel: !Ref EBServerApplicationVersion
```

Passes a value from the KeyName parameter

Links to the IAM instance profile created in the previous section

Start command

Passes the SQS queue into an environment variable

Passes the S3 bucket into an environment variable

Serves all files from /public as static files

Loads code from the book's S3 bucket

Under the hood, Elastic Beanstalk uses an ELB to distribute the traffic to the EC2 instances that are also managed by Elastic Beanstalk. You only need to worry about the configuration of Elastic Beanstalk and the code.

Elastic Beanstalk for the worker

The worker Elastic Beanstalk application is similar to the server. The differences are annotated in the following listing.

Listing 16.9 Imagery CloudFormation template: Elastic Beanstalk for the worker

```
EBWorkerApplication:
  Type: 'AWS::ElasticBeanstalk::Application'
  Properties:
    ApplicationName: 'imagery-worker'
    Description: 'Imagery worker: AWS in Action: chapter 16'
EBWorkerConfigurationTemplate:
  Type: 'AWS::ElasticBeanstalk::ConfigurationTemplate'
  Properties:
    ApplicationName: !Ref EBWorkerApplication
```

Describes the worker application container

```
    Description: 'Imagery worker: AWS in Action: chapter 16'
    SolutionStackName:
➥ '64bit Amazon Linux 2017.09 v4.4.0 running Node.js'
    OptionSettings:
    - Namespace: 'aws:autoscaling:launchconfiguration'
      OptionName: 'EC2KeyName'
      Value: !Ref KeyName
    - Namespace: 'aws:autoscaling:launchconfiguration'
      OptionName: 'IamInstanceProfile'
      Value: !Ref WorkerInstanceProfile
    - Namespace: 'aws:elasticbeanstalk:sqsd'
      OptionName: 'WorkerQueueURL'
      Value: !Ref SQSQueue
    - Namespace: 'aws:elasticbeanstalk:sqsd'
      OptionName: 'HttpPath'
      Value: '/sqs'
    - Namespace: 'aws:elasticbeanstalk:container:nodejs'
      OptionName: 'NodeCommand'
      Value: 'node worker.js'
    - Namespace: 'aws:elasticbeanstalk:application:environment'
      OptionName: 'ImageQueue'
      Value: !Ref SQSQueue
    - Namespace: 'aws:elasticbeanstalk:application:environment'
      OptionName: 'ImageBucket'
      Value: !Ref Bucket
EBWorkerApplicationVersion:
  Type: 'AWS::ElasticBeanstalk::ApplicationVersion'
  Properties:
    ApplicationName: !Ref EBWorkerApplication
    Description: 'Imagery worker: AWS in Action: chapter 16'
    SourceBundle:
      S3Bucket: 'awsinaction-code2'
      S3Key: 'chapter16/build/worker.zip'
EBWorkerEnvironment:
  Type: 'AWS::ElasticBeanstalk::Environment'
  Properties:
    ApplicationName: !Ref EBWorkerApplication
    Description: 'Imagery worker: AWS in Action: chapter 16'
    TemplateName: !Ref EBWorkerConfigurationTemplate
    VersionLabel: !Ref EBWorkerApplicationVersion
    Tier:
      Type: 'SQS/HTTP'
      Name: 'Worker'
      Version: '1.0'
```

Configures the HTTP resource that is invoked when an SQS message is received

Switches to the worker environment tier (pushes SQS messages to your app)

After all that YAML reading, the CloudFormation stack should be created. Verify the status of your stack:

```
$ aws cloudformation describe-stacks --stack-name imagery
{
  "Stacks": [{
    [...]
    "Description": "AWS in Action: chapter 16",
    "Outputs": [{
```

```
      "Description": "Load Balancer URL",
      "OutputKey": "EndpointURL",
      "OutputValue": "http://awseb-...582.us-east-1.elb.amazonaws.com"
    }],
    "StackName": "imagery",
    "StackStatus": "CREATE_COMPLETE"
  }]
}
```

Copy this output into your web browser.

Wait until CREATE_COMPLETE is reached.

The `EndpointURL` output of the stack contains the URL to access the Imagery application. When you open Imagery in your web browser, you can upload an image as shown in figure 16.17.

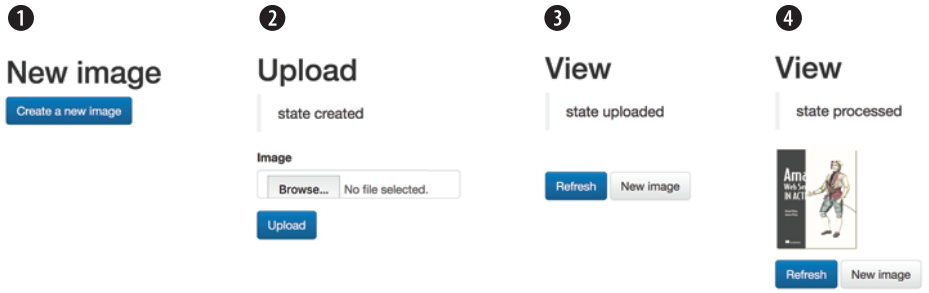


Figure 16.17 The Imagery application in action

Go ahead and upload some images and enjoy watching the images being processed.

Cleaning up

To get the name of the S3 bucket used by Imagery, run the following command in your terminal.

```
$ aws cloudformation describe-stack-resource --stack-name imagery \
➥ --logical-resource-id Bucket \
➥ --query "StackResourceDetail.PhysicalResourceId"
➥ --output text
imagery-000000000000
```

Delete all the files in your S3 bucket `imagery-000000000000`. Don't forget to replace *$bucketname* with the output from the previous command.

```
$ aws s3 rm s3://$bucketname --recursive
```

Execute the following command to delete the CloudFormation stack:

```
$ aws cloudformation delete-stack --stack-name imagery
```

Stack deletion will take some time.

Congratulations, you have accomplished a big milestone: building a fault tolerant application on AWS. You are only one step away from the end game, which is scaling your application dynamically based on load.

Summary

- Fault tolerance means expecting that failures happen, and designing your systems in such a way that they can deal with failure.
- To create a fault-tolerant application, you can use idempotent actions to transfer from one state to the next.
- State shouldn't reside on the EC2 instance (a stateless server) as a prerequisite for fault-tolerance.
- AWS offers fault-tolerant services and gives you all the tools you need to create fault-tolerant systems. EC2 is one of the few services that isn't fault-tolerant out of the box.
- You can use multiple EC2 instances to eliminate the single point of failure. Redundant EC2 instances in different availability zones, started with an auto-scaling group, are the way to make EC2 fault-tolerant.